Compacité pratique des diagrammes de décision valués. Normalisation, heuristiques et expérimentations

Hélène Fargier, Pierre Marquis, Nicolas Schmidt

Compacité pratique des diagrammes de décision valués : normalisation, heuristiques et expérimentations*

H. Fargier[1], P. Marquis[2], N. Schmidt[12]

[1] IRIT, 118 route de Narbonne, 31062 Toulouse Cedex
[2] CRIL, rue Jean Souvraz, 62307 Lens Cedex
{fargier,schmidt}@irit.fr marquis@cril.fr

Résumé

Les diagrammes de décision valués (VDDs) sont particulièrement intéressants pour la compilation de problèmes de satisfaction de contraintes valuées (VCSPs). L'intérêt des différents langages de la famille VDD (en particulier, les langages ADD, SLDD, AADD) est qu'ils admettent des algorithmes en temps polynomial pour des traitements (comme l'optimisation) qui ne sont pas polynomiaux à partir des VCSPs de départ. Comme l'efficacité pratique de tels traitements dépend de la taille du VDD compilée obtenu, il est important d'obtenir une forme la plus compacte possible. Nous décrivons dans cet article quelques résultats issus de nos travaux sur la compacité expérimentale des VDDs. Nous présentons un compilateur ascendant de VCSPs en SLDD$_+$ et SLDD$_\times$, un jeu d'heuristiques d'ordonnancement des variables, ainsi que des procédures de traduction des langages SLDD$_+$ et SLDD$_\times$ vers les langages ADD et AADD. Les différents langages cibles et les heuristiques ont été testés sur deux familles de jeux d'essai, des VCSPs additifs représentant des problèmes de configuration de voitures avec fonctions de coût, et des VCSPs multiplicatifs représentant des réseaux bayésiens. Il apparaît que, bien que le langage AADD soit strictement plus succinct en théorie que SLDD$_+$ et SLDD$_\times$, ces deux langages conviennent bien en pratique quand il s'agit de compiler des problèmes de nature purement additive (respectivement purement multiplicative).

Abstract

Valued decision diagrams (VDDs) prove valuable data structures for compiling valued constraint satisfaction problems (VCSPs). Indeed, languages from the VDD family (especially, ADD, SLDD, AADD) benefit from polynomial-time algorithms for some tasks of interest (e.g., the optimization one) for which no polynomial-time algorithm exists when the input is the VCSP considered at start. Since the practical efficiency of such tasks depends in practice on the size of the compiled VDD, it is important to look for diagrams which as as compact as possible. In this paper we present some results issued from our work on the practical compactness of VDDs. We present a VCSP-to-SLDD$_+$ bottom-up compiler and a VCSP-to-SLDD$_\times$ bottom-up compiler, several variable ordering heuristics, and some translation procedures from SLDD$_+$ and SLDD$_\times$ to ADD and to AADD. The target languages and the heuristics under consideration have been tested on two families of benchmarks, additive VCSPs representing car configuration problems with cost functions and multiplicative VCSPs representing Bayesian nets. It turns out that even if the AADD language is strictly more succinct (from the theoretical side) than any of SLDD$_+$ or SLDD$_\times$, those two languages perform well in practice when purely additive (respectively, purely multiplicative) problems are to be compiled.

*Ce travail a bénéficié du support du projet BR4CP ANR-11-BS02-008 de l'Agence Nationale de la Recherche.

1 Introduction

Les diagrammes de décision – automates, diagrammes de décision binaires ordonnés (OBDD), diagrammes de décision multivalués (MDD) – sont particulièrement intéressants pour la compilation de problèmes de satisfaction de contraintes, et spécialement pour le type d'application qui nous intéresse, à savoir la compilation de problèmes de configuration de produit [3, 10, 12]. Cela dit, ces diagrammes ne permettent pas tels quels de représenter des fonctions de coût, ou plus généralement des fonctions associant

une valuation (coût, degré de satisfaction, probabilité, etc.) aux affectations de variables – ils ne permettent pas la compilation de problèmes de satisfaction de contraintes *valuées* (VCSPs). Dans un tel cadre, les diagrammes de décision valués (VDDs) – diagrammes de décision algébriques (ADDs) [4], diagrammes de décision à arcs valués (EVBDDs) [15, 14, 3], *Semiring Labeled Decision Diagrams* (SLDDs) [20], diagrammes de décision algébriques affines (AADDs) [18][17] sont alors des langages cibles pertinents. Les ADDs par exemple ont été utilisés pour la compilation de problèmes de planification [13] ; des travaux en configuration de produit [3, 11] ont proposé d'utiliser les EVBDDs (ou de manière équivalente, les SLDDs additifs – SLDD_+) pour capturer des fonctions de coût ou de préférences fortement additives, c'est-à-dire des cas où la valuation associée à une affectation des variables s'exprime directement par un VCSP dont toutes les contraintes souples sont unaires : l'idée est alors de compiler les contraintes dures qui définissent le produit configurable par un MDD, puis d'ajouter les valuations définies par les contraintes unaires directement sur les arcs.

L'intérêt de ces langages de compilation est qu'ils permettent une manipulation efficace de l'ensemble des solutions du VCSP, une fois compilé. La résolution interactive (c'est-à-dire par l'utilisateur) d'un problème de configuration avec fonction coût par exemple se rénduit en effet à des opérations de conditionnement, de propagation et d'optimisation qui sont linéaires dans la taille de la structure compilée. Ce qui est donc important en pratique, c'est d'obtenir une forme compilée la plus compacte possible. Plusieurs facteurs influencent cette compacité :

- la compacité théorique (ou "*succinctness*") qui est relative (elle indique s'il est possible de séparer exponentiellement ou non un langage d'un autre) et qui se focalise sur le pire cas ;
- la canonicité, c'est-à-dire la capacité de chaque langage à offrir pour chaque VCSP une forme canonique – ceci permet de reconnaître et de fusionner efficacement (par exemple, par un mécanisme de cache) les sous-diagrammes équivalents ;
- les heuristiques d'ordonnancement des variables choisies pour construire le diagramme de décision.

Les travaux de Sanner et Mc Allester ont montré que le langage AADD offre une forme canonique et est plus performant que le langage ADD du point de vue de la compacité théorique comme du point de vue pratique. Dans des travaux récents [**?**], nous avons montré que la propriété de canonicité à l'œuvre dans les AADD pouvaient être étendue, algébriquement, au langage SLDD, et qu'en théorie au moins, le langage AADD est plus succinct que SLDD_+ (resp. $\text{SLDD}_\times$), c'est-à-dire le langage des SLDDs fondé sur le semi-anneau commutatif $\langle \mathbb{R}^+ \cup \{+\infty\}, 0, +\infty, +, \min \rangle$ (resp. sur le semi-anneau commutatif $\langle \mathbb{R}^+, 1, 0, \times, \max \rangle$).

Nous décrivons dans la suite quelques résultats issus de nos travaux sur la compacité expérimentale des VDDs. Nous présentons un compilateur ascendant de VCSPs en SLDD_+ et $\text{SLDD}_\times$, un jeu d'heuristiques d'ordonnancement des variables, ainsi que des procédures de traduction des langages SLDD_+ et $\text{SLDD}_\times$ vers les langages ADD et AADD. Les différents langages cibles et les heuristiques ont été testés sur deux familles de jeux d'essai, des VCSPs additifs représentant des problèmes de configuration de voitures avec fonctions de coût, et des VCSPs multiplicatifs représentant des réseaux bayésiens. Il apparaît que, quoique le langage AADD soit strictement plus succinct en théorie que SLDD_+ et $\text{SLDD}_\times$, ces deux langages conviennent bien en pratique quand il s'agit de compiler des problèmes de nature purement additive (respectivement purement multiplicative).

2 Diagrammes de décision valués

Soit $X = \{x_1, \ldots, x_n\}$ un ensemble de variables où chaque $x_i \in X$ prend ses valeurs dans un domaine discret D_x ; on note D_X l'ensemble des affectations $\vec{x}$ de X. Un diagramme de décision α est une structure de données permettant de représenter une fonction f_α qui associe à chaque affectation $\vec{x} = \{(x_i, d_i) \mid d_i \in D_{x_i}, i = 1, \ldots, n\}$ un élément d'un ensemble E de valuations. E est à la base d'une structure de valuation $\mathcal{E}$ qui peut être plus ou moins riche d'un point de vue algébrique. Dans le formalisme ADD, aucune hypothèse n'est faite sur E (bien que l'on considère généralement que $E = \mathbb{R}$). Pour le langage AADD, $E = \mathbb{R}^+$.

Définition 1 *Un* diagramme de décision valué *(VDD) est un graphe orienté et acyclique avec une seule racine, où chaque nœud N est étiqueté par une variable $x \in X$: si $D_x = \{d_1, \ldots, d_k\}$, alors N a k arcs sortants $a_1, \ldots, a_k$, tels que chaque a_i est valué par $val(a_i) = d_i$.*
Les variables étiquetant les nœuds de tout chemin de la racine à une feuille sont toutes distinctes. Les nœuds N (resp. les arcs a) peuvent également être étiquetés par une valeur $\phi(N)$ (resp. $\phi(a)$) de E. On note $In(N)$ (respectivement $Out(N)$) les arcs entrants dans (respectivement issus de) N.

Les diagrammes de décision valués sont généralement *ordonnés* : un ordre total $<$ sur X est choisi de manière à ce que la suite des variables associées aux nœuds rencontrés sur chaque chemin de la racine vers une feuille soit compatible avec cet ordre.

Un diagramme de décision valué est dit *sous forme réduite* s'il ne contient pas de nœuds isomorphes (deux nœuds étiquetés par la même variable et dont les arcs sortants sont identiques, c'est-à-dire pointent sur les mêmes nœuds en portant la même valeur du domaine de leur variable et la même valuation ϕ). Tout VDD possède une unique forme réduite, qu'il est possible d'obtenir en temps linéaire en sa taille, soit par une procédure de réduction remontant de la/des feuille/s vers la racine, soit simplement par un mécanisme de cache (une "*unique table*"). Dans la suite, nous supposons que les diagrammes de décision considérés sont sous forme réduite.

Les ADDs sont la généralisation aux valuations non booléennes des OBDDs, les deux nœuds terminaux *true* et *false* étant remplacés par autant de nœuds que de valeurs de E associées à une affectation au moins.

Définition 2 *Un ADD est un VDD ordonné dont seuls les nœuds terminaux sont valués (les arcs ne le sont pas). Un ADD α associe à chaque affectation $\vec{x} \in D_X$ la valeur $f_\alpha(\vec{x}) \in E$ définie par :*

- *si α est un nœud terminal N étiqueté par un élément $\phi(N)$ de E, alors $f_\alpha(\vec{x}) = \phi(N)$;*
- *sinon, la racine N de α est étiquetée par $x \in X$; soient d la valeur de x dans $\vec{x}$, $a = (N, M)$ l'arc issu de N tel que $v(a) = d$, et β le ADD de racine M dans α ; on a pour tout $\vec{x} \in D_X$: $f_\alpha(\vec{x}) = f_\beta(\vec{x})$.*

Les nœuds terminaux reprenant l'ensemble des valeurs possibles de la fonction représentée, le nombre de nœuds d'un ADD croît avec le cardinal de l'ensemble de ces valeurs (l'image de la fonction représentée). Ainsi, la fonction $f(x_1, \ldots, x_n) = \Sigma_{i=1}^{n} 2^{i-1} x_i$ sur $\{0,1\}^n$, qui est représentable en espace polynomial par un VCSP fortement additif, prend 2^n valeurs différentes, d'où une taille exponentielle pour les ADDs qui la représentent.

Dans les SLDDs (*Semiring Labeled Decision Diagrams*) tels que définis dans [20], la structure de valuation doit être un semi-anneau $\mathcal{E} = \langle E, \otimes, \oplus, 1_s, 0_s \rangle$ – 1_s dénotant l'élément neutre de l'opérateur $\otimes$ et 0_s dénotant l'élément neutre de l'opérateur $\oplus$, absorbant pour $\otimes$. L'opérateur $\oplus$ n'a aucune influence pour la définition d'un SLDD en tant que représentation d'une fonction de D_X dans E ($\oplus$ est utilisé lorsque l'on veut calculer une valuation optimale, ou lorsque l'on veut éliminer une ou plusieurs variables). Pour cette raison, nous utilisons dans la suite une définition un petit peu plus générale que celle de [20], exigeant simplement une structure de monoïde pour $\mathcal{E} = \langle E, \otimes, 1_s \rangle$: $\otimes$ est une loi interne à E, associative, et qui possède un élément neutre 1_s.

Définition 3 *Un SLDD α sur X est un VDD avec une unique racine et un unique nœud terminal, dont les arcs sont étiquetés par des éléments de E où $\mathcal{E} = \langle E, \otimes, 1_s \rangle$ est un monoïde. Un SLDD associe à chaque affectation $\vec{x} \in D_X$ la valeur $f_\alpha(\vec{x})$ appartenant à E définie par :*

- *si α est le nœud terminal alors $f_\alpha(\vec{x}) = 1_s$;*
- *sinon, la racine N de α est étiquetée par $x \in X$; soient $d \in D_x$ la valeur de x dans $\vec{x}$, a l'arc issu de N tel que $v(a) = d$, M son extrémité et β le SLDD de racine M dans α ; on a pour tout $\vec{x} \in D_X$: $f_\alpha(\vec{x}) = \phi(a) \otimes f_\beta(\vec{x})$.*

À des fins de normalisation, on peut associer à α une valeur $\phi_0 \in E$ (son "offset"). La fonction "augmentée" que représente α est définie par, pour tout $\vec{x} \in D_X$, $f_{\alpha,\phi_0}(\vec{x}) = \phi_0 \otimes f_\alpha(\vec{x})$.

Deux monoïdes sont particulièrement intéressants : $\mathcal{E} = \langle \mathbb{R}^+ \cup \{+\infty\}, +, 0 \rangle$ pour tous les problèmes dont les valuations sont de nature additive (coûts) et $\mathcal{E} = \langle \mathbb{R}^+, \times, 1 \rangle$ pour tous les problèmes dont les valuations sont de nature multiplicative (probabilités). Les langages associés sont notés respectivement SLDD_+ et $\text{SLDD}_\times$. Chacun admet un élément absorbant (0 pour $\text{SLDD}_\times$, $+\infty$ pour les SLDD_+), ce qui permet de compiler des VCSPs possédant des contraintes dures : dans un SLDD_+ par exemple, toute affectation $\vec{x}$ telle que $f(\alpha)(\vec{x}) = +\infty$ est considérée comme non admissible car violant une contrainte dure.

Enfin, les diagrammes de décision algébriques affines introduits dans [18][17] permettent d'utiliser conjointement les opérateurs $\times$ et $+$ sur $\mathbb{R}^+$. Dans un SLDD, chaque arc a porte une valeur $\phi(a)$; dans un AADD, les arcs sont étiquetés par des *couples* de valeurs.

Définition 4 *Un AADD α sur X est un VDD ordonné avec une unique racine et un unique nœud terminal, dont les arcs sont étiquetés par des couples d'éléments de $\mathbb{R}^+$. α associe à chaque affectation $\vec{x} \in D_X$ la valeur $f_\alpha(\vec{x}) \in \mathbb{R}^+$ définie par :*

- *si α est le nœud terminal N, $f_\alpha(\vec{x}) = 1$;*
- *sinon, la racine N de α est étiquetée par $x \in X$; soient $d \in D_x$ la valeur de x dans $\vec{x}$, a l'arc issu de N tel que $v(a) = d$, M son extrémité, $\phi(a) = \langle q, f \rangle$ le couple de valeurs associée à a et β le AADD de racine M dans α ; on a pour tout $\vec{x} \in D_X$: $f_\alpha(\vec{x}) = q + f \times f_\beta(\vec{x})$.*

À des fins de normalisation, on attache à α un couple $\langle q_0, f_0 \rangle$ de $\mathbb{R}^+ \times \mathbb{R}^+$ (son "offset"). La fonction "augmentée" que représente α est définie par, pour tout $\vec{x} \in D_X$, $f_{\alpha,\langle q_0,f_0 \rangle}(\vec{x}) = q_0 + f_0 \times f_\alpha(\vec{x})$.

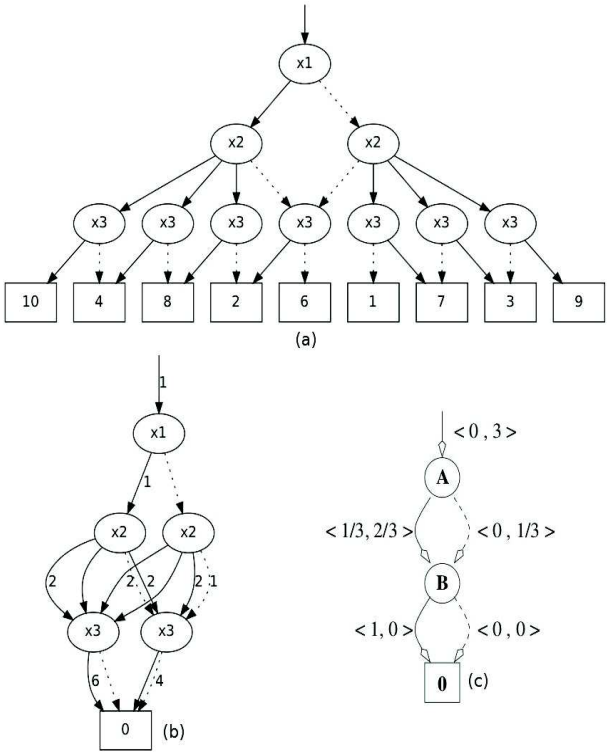

FIGURE 1 – (a) : Exemple de ADD ; (b) Exemple de $SLDD_+$; (c) Exemple de AADD

Un $SLDD_+$ peut être considéré comme un AADD particulier [1] dont les facteurs multiplicatifs portés par les derniers arcs (aboutissant au puits) sont égaux à 0 alors que tous les autres sont égaux à 1. De la même façon, un $SLDD_\times$ peut être considéré comme un AADD particulier dont les facteurs additifs sont nuls. Dans la suite, nous supposerons les SLDDs ordonnés selon un ordre $<$ sur X fixé ; cela est nécessaire d'une part pour pouvoir les comparer aux AADDs et aux ADDs qui sont des structures ordonnées, et d'autre part parce que cette propriété garantit une forme réduite canonique (et des opérations de combinaison en temps polynomial). La figure 2 décrit (a) un ADD, (b) un $SLDD_+$ et (c) un AADD. L'ADD et le $SLDD_+$ représentent la même fonction ; l'AADD représentant cette fonction serait très similaire au $SLDD_+$ décrit en (b). L'AADD donné à la figure 2 (c) représente la fonction $f = a + (a + 1)b$.

3 Normalisation

Le langage AADD est muni d'une procédure de normalisation qui permet de garantir, pour chaque fonction à représenter et étant donné un ordre des variables, une unique représentation réduite. Cette propriété joue un rôle important dans l'obtention de forme compacte, puisqu'elle permet de détecter, et donc de fusionner, les sous-diagrammes qui représentent la même sous-fonction.

1. En étendant E à $(\mathbb{R}^+ \cup \{+\infty\}) \times \mathbb{R}^+$.

Algorithme 1 : normalizeAADD$(\alpha, \langle q_0, f_0 \rangle)$

input : Un AADD α et son "offset" $\langle q_0, f_0 \rangle$
output : Un AADD normalisé équivalent à α

for *each node* N *of* α *in inverse topological ordering* do
  $q_{min} \leftarrow \min_{a \in Out(N)} q_a$;
  $range \leftarrow \max_{a \in Out(N)}(q_a + f_a) \; - \; q_{min}$;
  for *each* $a \in Out(N)$ do
    if $range > 0$ then
      $q_a \leftarrow (q_a - q_{min})/range$;
      $f_a \leftarrow f_a/range$;
    else `// here,` $f_a = 0$ `and` $q_a = q_{min}$
    $q_a \leftarrow q_a - q_{min}$;
  for *each* $a \in In(N)$ do
    $q_a \leftarrow q_a + q_{\min} \times range$;
    $f_a \leftarrow f_a \times q_{range}$;
$q_0 \leftarrow q_0 + q_{min} \times range$;
$f_0 \leftarrow f_0 \times range$;
return α;

3.1 Normalisation des AADD [17]

Soient un nœud N étiqueté par une variable x dont le domaine contient n valeurs, $Out(N) = \{a_1, a_2, \ldots, a_n\}$ l'ensemble des arcs sortants de N, et pour chaque $i \in 1, \ldots, n$, $\phi(a_i) = \langle q_i, f_i \rangle$ la valuation attachée à cet arc. N est *normalisé* ssi :

- $min(q_1, q_2, ..., q_n) = 0$;
- $max(q_1 + f_1, q_2 + f_2, ..., q_n + f_n) = 1$;
- le sous-AADD β sur lequel pointe un a_i est tel que si $\forall \vec{x}, f(\beta)(\vec{x}) = 0$, alors $f_i = 0$.

Un AADD est normalisé ssi tous ses nœuds le sont.

Tout AADD peut être normalisé en temps linéaire (voir algorithme 1). Cette procédure normalise les nœuds du puits vers la racine ; pour chaque N, elle calcule un "offset interne" $\langle q_{min}, range \rangle$, qui permet de normaliser N et puis est reporté sur les parents de N, ou sur l'offset de l'AADD si N est sa racine. Appliquée depuis le puits vers la racine, cette procédure remonte sur l'offset du diagramme les valeurs minimum (q_0) et maximum ($q_0 + f_0$) de la fonction représentée par l'AADD.

Les $SLDD_+$ et $SLDD_\times$ pouvant être vus comme des AADDs, nous pouvons nous appuyer sur l'existence d'une forme normalisée de ces structures pour proposer une définition similaire allégée de leur forme normalisée :

Algorithme 2 : normalizeSLDD(α, ϕ_0)

input : Un SLDD α, et son "offset" ϕ_0
output : Un SLDD normalisé équivalent à α
// For $SLDD_+$ let
// $\otimes = +, \otimes^{-1} = -, \oplus = min, 1_s = 0$
// For $SLDD_\times$ let
// $\otimes = \times, \otimes^{-1} = \oplus = max, 1_s = 1$
for *each node* N *of* α *in inverse topological ordering* do
  $\phi_m \leftarrow \oplus_{a \in Out(N)} \phi(a)$;
  // Rem: when $\oplus = max$, $\phi_m = 0$ iff
  // $\forall a, \phi(a) = 0$
  for *each* $a \in Out(N)$ do
    if $\phi(a) = \phi_m$ then $\phi(a) \leftarrow 1_s$;
    else $\phi(a) \leftarrow \phi(a) \otimes^{-1} \phi_m$;
  for *each* $a \in In(N)$ do
    $\phi(a) \leftarrow \phi(a) \otimes \phi_m$;
$\phi_0 \leftarrow \phi_0 \otimes \phi_m$;
return α;

Définition 5

- *Un $SLDD_+$ est normalisé ssi pour tout nœud N d'arcs sortants $Out(N) = \{a_1, a_2, \ldots, a_n\}$, $min(\phi(a_1), \phi(a_2), ..., \phi(a_n)) = 0$.*
- *Un $SLDD_\times$ est normalisé ssi pour tout nœud N d'arcs sortants $Out(N) = \{a_1, a_2, \ldots, a_n\}$, $max(\phi(a_1), \phi(a_2), ..., \phi(a_n)) = 1$.*

De même, on peut utiliser une spécialisation de la procédure de [17] pour normaliser un SLDD (voir algorithme 2). Dans un $SLDD_+$ (resp. $SLDD_\times$) normalisé, le minimum (resp. le maximum) de la fonction représentée est ainsi remonté à la racine en offset.

4 Construction de diagrammes de décision valués

À des fins expérimentales, nous avons implémenté un compilateur de VDDs. Ce compilateur permet de compiler d'une part des VCSPs à valuations additives [5] dont les contraintes sont exprimées par des tables tuple↦valuation (selon le format XCSP 2.1 décrit dans [16]) sous la forme de $SLDD_+$; et d'autre part de compiler des réseaux bayésiens sous la forme de $SLDD_\times$ (selon le format XML de [7]). Pour permettre l'obtention de différents types de VDDs, nous avons également implémenté des traductions de $SLDD_+$ et $SLDD_\times$ vers ADD et réciproquement, ainsi que des traductions de $SLDD_+$ et $SLDD_\times$ vers AADD.

4.1 Compilation

Notre approche de la compilation d'un VCSP en SLDD suit un procédé ascendant classique pour la construction de diagrammes de décisions ordonnés, valués ou non [17][6][2]. Nous décrivons ici la procédure d'ajout d'une contrainte valuée à un $SLDD_+$. L'ajout d'une contrainte C à un $SLDD_\times$ suit le même schéma, en remplaçant toute addition par une multiplication, toute soustraction par une division, et toute occurrence de l'élément neutre 0 par une occurrence de l'élément neutre 1.

On détermine tout d'abord un ordre total $<$ des variables selon lesquels le SLDD à construire sera ordonné (la section suivante décrit et compare les heuristiques d'ordonnancement de variables). On crée ensuite un SLDD "blanc ", c'est-à-dire que l'on associe à chaque variable un nœud dont chacun des arcs sortants (il y en a un par valeur dans le domaine de la variable) pointe vers le nœud portant la variable suivante selon l'ordre $<$ avec une valuation $\phi = 0$ (ou $\phi = 1$ pour un $SLDD_\times$). Les contraintes sont ajoutées une à une au SLDD en construction.

Dans un premier temps, lorsque la contrainte à ajouter contient un ϕ_0 par défaut (à appliquer à toutes les affectations qui ne sont pas explicitement dans la table de la contrainte), on met à jour l'étiquette de tout arc a sortant d'un nœud étiqueté par la dernière variable de C (selon $<$) via $\phi(a) \leftarrow \phi(a) + \phi_0$ (cf. algorithme 3). On ajoute ensuite chacun des tuples t de la table au SLDD courant, selon une procédure inspirée de la procédure $Apply(+)$ décrite dans [17] [2] – voir algorithme 4. Le SLDD résultant est normalisé et réduit, et devient le nouvel SLDD courant.

Algorithme 3 : AddConstraint(C, ϕ_0, α)

input : Un SLDD ordonné α, une contraint C, une valuation par defautϕ_0
output : Le SLDD α auquel on a ajouté la contrainte C

$x \leftarrow last(Var(C))$ //dernière var. de C selon $<$
for *all* a *sortant d'un nœud étiqueté par* x do
  $\phi(a) \leftarrow \phi(a) + \phi_0$
$y \leftarrow first(Var(C))$ //première var. de C selon $<$
for $(t, \phi_t) \in Table(C)$ do
  for *nœud* $N \in \alpha$ *t.q.* $Var(N) = y$ do
    $AddTuple(t, \phi_t, N)$
$reduce(normalizeSLDD(\alpha))$

2. Cette procédure étant elle-même une extension de la procédure de conjonction de OBDDs proposée dansr [6] au cas d'autres opérateurs ($\times$, $+$, min, *etc.*) que la conjonction.

Algorithme 4 : AddTuple(t, ϕ_t, α)

```
input  : Un SLDD ordonné α, un n-uplet t, une
         valuation φ_t
output : Le SLDD α auquel on a ajouté le tuple t
// ⊗ = × for SLDD_×; ⊗ = + for SLDD_+
x ← Var(N);
if x ∈ Var(t) then
    let a = (N, M) be the arc in Out(N)
    corresponding to value t[x];
    if x = last(Var(t)) then
        φ(a) ← φ(a) ⊗ φ_t ;
    else
        if M_t = null then
            M_t ← clone(M) // same successors
            // as those of M, reached by
            // copies of the arcs in Out(N)
            Out(a) ← M_t ;
        AddTuple(t, φ_t, M_t);
else
    for a = (N, M) ∈ Out(N) do
        if M_t = null then
            M_t ← clone(M) ;
            Out(a) ← M_t ;
        AddTuple(t, φ_t, M_t) ;
```

4.2 Heuristiques

La taille des diagrammes de décision classiques est évidemment très sensible à l'ordre dans lequel ses variables ont été ordonnées [2][8] – et c'est également le cas pour leurs versions valuées. Nous avons appliqué aux VDDs plusieurs heuristiques proposées pour les MDDs [2][8].

MCF [2] Afin de réduire la taille du diagramme de décision, il peut sembler intéressant de rencontrer les variables les plus contraintes (intervenant dans le plus de contraintes) le plus rapidement possible. On espère ainsi, lorsque le problème contient des contraintes dures, poster le plus tôt possible des arcs portant la valeur absorbante (n-uplet non admissible) – ces arcs joignant directement le puits, cela limite la taille de du VDD. L'heuristique MCF (pour *Most Constrained First*) trie les variables en fonction du nombre de contraintes dures dans lesquelles elles sont utilisées. $MCF(x) = |\{c \mid x \in Var(c)\}|$.

Band-Width [2] a montré que la *Band-Width* du graphe de contraintes permet de borner la taille du meilleur MDD décrivant un CSP classique. Soit $G = (X, C)$ un graphe de contraintes et $O : \{1, \ldots, n\} \mapsto X$ un ordre sur les n variables de X (O associe à chaque rang une variable). La *Band-Width* d'un ordre O est : $BW(O) = max\{j - i$ t.q. $i < j$ et $\exists$C, $O[i], O[j] \in Var(C)\}$.
Pour un O donné, on place en tête les variables de plus fort BW dans l'ordre.

Le calcul d'un ordre de *Band-Width* minimum pour un graphe quelconque étant un problème NP-complet, nous utilisons un algorithme glouton pour l'approcher. On choisit les variables de l'ordre itérativement : la première sélectionnée (celle qui sera en tête du VDD) est la variable la plus contrainte. Étant donnée une suite O de k variables sélectionnées, la variable suivante est la variable x non encore sélectionnée qui maximise la quantité : $H_O(x) = max\{|O| - i \mid O[i]$ voisine de $x\}$.

MCS-Inv [19] MCS (pour *Maximum Cardinal Search*) est une méthode introduite dans le cadre de reconnaissance de graphes triangulés. La méthode MCS-Inv reprend le même principe en inversant l'ordre final. Elle permet aux variables fortement contraintes d'être proches des variables avec lesquelles elles sont liées. Notre implémentation utilise un algorithme glouton. On choisit les variables de l'ordre itérativement : la première sélectionnée est la variable la plus contrainte. Étant donnée une suite O de k variables sélectionnées, la variable suivante est la variable x non encore sélectionnée qui maximise la quantité $H_O(x) = \Sigma_{i=1,\ldots,k,O[i]}$ voisine de $_x|S| - i$.

Force [1] Le but de *Force* est de minimiser le *span* induit sur le graphe de contraintes, c'est-à-dire la somme (et non pas, comme pour l'heuristique *Band-Width*, le maximum) des distances séparant les variables appartenant à la même contrainte. $span(O) = \Sigma_C(j - i / i < j$ et $O[i], O[j] \in Var(C))$.
La méthode consiste à calculer, à partir d'un ordre quelconque, le "centre de gravité (COG)" de chacune des contraintes C (en fonction de la position des variables dans l'ordre).

$$COG(C) = \frac{\Sigma_{x \in Var(C)} POS(x)}{|Var(C)|}.$$

On remet à jour ensuite les positions de chacune des variables en fonction des centres de gravité de l'ensemble des contraintes auxquels elles appartiennent.

$$POS(x) = \frac{\Sigma_{C, x \in Var(C)} COG(c)}{|\{C, x \in Var(C)\}|}.$$

Cette procédure part d'un ordre quelconque sur les variable O : à l'initialisation, $POS(x)$ est le rang de x dans O. Elle est répétée autant de fois que nécessaire, jusqu'à arriver à un point fixe, où, d'une itération à l'autre, les estimations de centres de gravités des va-

riables n'évoluent plus. On réordonne ensuite les variables par POS croissant.

4.3 Traductions ADD⇌SLDD⇌AADD

Nous développons ici les principes qui fondent les traductions d'un type de VDD vers chacun des autres (en supposant que tous partagent le même domaine de valuation, typiquement $E = \mathbb{R}^+ \cup \{+\infty\}$). Ces traductions peuvent souvent être vues comme l'application de procédures de normalisation.

Traductions SLDD↦AADD, ADD↦SLDD, ADD↦AADD

La traduction la plus évidente est la traduction d'un SLDD_+ (ou d'un $\text{SLDD}_\times$) en AADD : tout SLDD_+ peut être transformé en AADD en remplaçant, pour chaque arc a à destination du puits son étiquette $\phi(a)$ par le couple $\langle \phi(a), 0 \rangle$, et pour tout autre arc a', son étiquette $\phi(a')$ par le couple $\langle \phi(a'), 1 \rangle$. On normalise ensuite l'AADD obtenu, ce qui permet éventuellement de le réduire et d'obtenir une structure de plus faible taille. De la même façon, tout $\text{SLDD}_\times$ peut être transformé en AADD en remplaçant, pour chaque arc a, son étiquette $\phi(a)$ par le couple $\langle 0, \phi(a) \rangle$. On normalise et réduit ensuite l'AADD obtenu, ce qui permet éventuellement d'obtenir une structure plus compacte que le diagramme original.

De la même façon, on peut toujours transformer un ADD en SLDD_+ ou en $\text{SLDD}_\times$ en reportant les valuations portées par les nœuds terminaux sur leurs arcs entrants (les autres arcs portant la valuation $\phi = 0$ lorsque l'on veut construire un SLDD_+ , la valuation $\phi = 1$ lorsque l'on veut construire un $\text{SLDD}_\times$) ; les nœuds terminaux sont remplacés par le puits du nouveau SLDD. On normalise et réduit ensuite le diagramme obtenu selon les principes de normalisation des SLDD_+ (respectivement des $\text{SLDD}_\times$).

La procédure est la même lorsqu'il s'agit de transformer un ADD en AADD, les valuations ϕ_i des nœuds terminaux étant reportées sur leurs arcs entrants sous la forme d'un couple $\langle 0, \phi_i \rangle$.

Traductions SLDD↦ADD, AADD↦ADD, AADD↦SLDD

Transformer un SLDD en ADD revient à repousser les valuations ϕ vers les arcs du dernier niveau. En quelque sorte, il s'agit d'une normalisation assurant que pour tout a, $\phi(a)$ est égal à l'élément neutre (0 pour les SLDD_+ et 1 pour les $\text{SLDD}_\times$) ; il faut alors, pour porter les valuations, dupliquer le puits en autant de nœuds finaux que de valuations différentes sur ses arcs entrants. La procédure de "normalisation" du

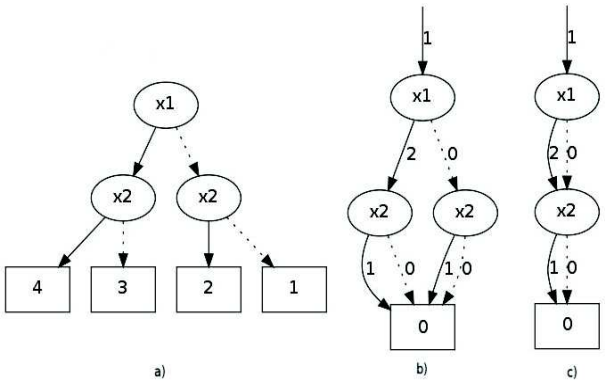

FIGURE 2 – Exemple de traduction d'un ADD (a) en SLDD_+ : en (b) les poids sont remontés sur les arcs et les nœuds sont normalisés ; en (c) les nœuds isomorphes sont fusionnés

SLDD en ADD procède de la racine vers le puits ; plus précisément, pour tout nœud N dont les parents ont été traités :

- remplacer N par autant de copies $N_1, \ldots N_k$ de N que de valuations différentes $\phi_1, \ldots, \phi_k$ sur les arcs entrant dans N ;
- si N est le puits du diagramme de décision, chaque copie N_i porte la valuation ϕ_i ;
- sinon (N est un nœud interne), pour chaque copie N_i, et chaque arc a de N vers un nœud M, créer un arc a' étiqueté $Val(a)$ de N_i vers M, prenant pour valeur $\phi(a') = \phi(a) + \phi_i$;
- normaliser et réduire le diagramme en fusionnant les nœuds isomorphes.

Le même type de procédure est appliqué pour transformer un AADD en SLDD_+, c'est-à-dire en AADD dont les valuations sont de la forme $\langle q, 1 \rangle$: on crée autant de copies $N_1, \ldots, N_k$ de N que de facteurs multiplicatifs différents $f_1, \ldots, f_k$ sur les arcs a_j entrant dans N. Chaque copie N_i est liée aux prédécesseurs de N par des copies a' des arcs a entrant dans N et portant la valuation f_i : chaque arc a' porte la même valeur du domaine de sa variable que a et sa valuation est $\langle q_a, 1 \rangle$; le facteur multiplicatif de a est reporté sur les arcs suivants : pour chaque copie N_i, et chaque arc a de N vers un nœud M, étiqueté par $\langle q, f \rangle$, on crée un arc a' étiqueté $Val(a)$ de N_i vers M, prenant pour valeur $\langle q' \times f_i, f' \times f_i \rangle$.

Pour obtenir un $\text{SLDD}_\times$ plutôt qu'un SLDD_+, on "normalise" le diagramme affine de manière à assurer que toutes les valuations sont de la forme $\langle 0, f \rangle$. Pour cela, on crée autant de copies $N_1, \ldots, N_k$ de N que de facteurs $q_1 + f_1, \ldots, q_k + f_k$ différents sur les arcs a_j entrant dans N. Chaque copie a' d'un $a \in In(N)$ porte la valuation $\langle 0, f_a \rangle$; le facteur additif est reporté sur

TABLE 1 – Comparaison des heuristiques MCF, *Band-Width*, MCS-Inv et *Force*

	MCF		*Band-Width*		MCS-Inv		*Force*	
Instance	nœuds(arcs)	tps(s)	nœuds(arcs)	tps(s)	nœuds(arcs)	tps(s)	nœuds(arcs)	tps(s)
VCSP↦SLDD$_+$								
SmallPriceOnly	105(239)	< 1	40(120)	< 1	36(108)	< 1	351(2946)	< 1
MedPriceOnly	777(2009)	< 1	312(906)	< 1	169(499)	< 1	3362(40474)	1
BigPriceOnly	-	t-o	97646(251594)	4002	3317(9687)	18	8873 (824180)	499
Small	3100(7564)	1	4349(10451)	2	2344(5584)	1	4503 (12430)	2
Medium	5660(19363)	11	11700(30835)	18	6234(17062)	7	13603(34263)	32
Big	-	t-o	-	t-o	198001(925472)	79043	-	t-o
Bayes↦SLDD$_\times$								
Cancer	13(25)	<1	13(25)	<1	13(25)	<1	13(25)	<1
Asia	36(71)	<1	29(57)	<1	23(45)	<1	25(49)	<1
Car-starts	77(157)	<1	41(83)	<1	41(83)	<1	49(99)	<1
Alarm	-	t-o	5852(14899)	<1	1301(3993)	<1	7054(20134)	<1
Hailfinder25	-	t-o	-	t-o	32718(108083)	8	m-o	-

les arcs suivants : pour chaque copie N_i, et chaque arc a de N vers un nœud M, étiqueté par $\langle q, f\rangle$, on crée un arc a' étiqueté $Val(a)$ de N_i vers M, prenant pour valeur $\langle q' + \frac{q_i}{f_i}, f'\rangle$. Finalement, les étiquettes des arcs entrant dans le puits sont mises sous la forme $\langle q+f, 0\rangle$.

Enfin, pour obtenir un ADD plutôt qu'un SLDD à partir d'un AADD, on crée autant de copies N_i de N que de valeurs $q_i + f_i$ différentes sur les arcs entrant dans N et le facteur $q_i + f_i$ est repoussé sur les arcs sortant des N_i : les étiquettes $\langle q', f'\rangle$ des arcs a' deviennent $\langle q_i + f_i \times q', f_i \times f'\rangle$.

Notons que ces transformations peuvent faire "exploser" le diagramme, ce qui est inévitable dans le pire cas : la fonction $f(x_1, \ldots, x_n) = \Sigma_{i=1}^{n} 2^{i-1} x_i$ sur $\{0,1\}^n$ par exemple, prend 2^n valeurs différentes ; sa représentation par un ADD posséde donc 2^n feuilles, alors qu'elle peut être représentée par un SLDD$_+$ (et donc par un AADD) à $n+1$ nœuds et $2n$ arcs. On peut également montrer que cette fonction ne peut pas être représentée par un SLDD$_\times$ de taille polynomiale. Formellement, le langage AADD est strictement plus succinct que celui des SLDD$_+$, lui-même strictement plus succinct que ADD. Il est également strictement plus succinct que le langage SLDD$_\times$ lui-même strictement plus succinct que ADD [9].

Le nombre d'opérations effectuées lors de la transformation en AADD et d'un AADD vers un autre langage augmente le risque d'erreurs d'arrondis (on trouve en effet plus facilement dans les AADD des additions entre deux valeurs d'ordres de grandeur différents, dont le résultat sera nécessairement arrondi, oubliant la valeur la plus faible). C'est pourquoi dans les faits, nous évitons les traductions de AADD vers SLDD et ADD. Par exemple, transformer un SLDD$_+$ en SLDD$_\times$ (ou inversement), on passe par le langage ADD et non AADD. Même si le passage en ADD n'est pas toujours possible (à cause de l'explosion en espace), cela évite d'obtenir un résultat erroné. Le passage de SLDD$_\times$ vers SLDD$_+$ (en passant par un ADD) peut aussi entraîner des erreurs d'arrondis s'il y a de trop gros écarts d'ordres de grandeur en sortie d'un SLDD$_\times$.

5 Résultats expérimentaux

Dans cette section, nous comparons expérimentalement, d'une part, l'efficacité des différentes heuristiques, et, d'autre part, la compacité pratique des différents types de VDDs décrits dans les sections précédentes.

Nous avons testé nos structures de données et heuristiques sur deux familles de jeux d'essai, des VCSP additifs (codant des problèmes de configuration de véhicule) et des VCSP multiplicatifs (codant des réseaux bayésiens). En ce qui concerne les réseaux bayésiens, nous avons utilisé des jeux d'essai standard au format XML [7]. Les instances de CSP pondérés représentant des problèmes de configuration de voitures nous ont été fournies par Renault – elles sont disponibles via [**?**]. Ces instances sont composées de contraintes dures, définissant les modèles de voitures faisables (la diversité de la gamme, en termes automobiles) ainsi que de contraintes valuées représentant les coûts, le prix d'un véhicule étant la somme des coût spécifiés par les différentes contraintes valuées. Trois jeux d'essai nommés **Small**, **Medium** et **Big** représentent trois modèles différents de voitures (deux citadines et un utilitaire). Les caractéristiques des jeux d'essai sont les suivantes :

- **Small** : #variables=139 ; #taille du domaine max=16 ; #contraintes=176 (incluant 29 de coût)
- **Medium** : #variables=148 ; #taille du domaine max=20 ; #contraintes=268 (incluant 94 de coût)
- **Big** : #variables=268 ; #domaine max=324 ; #contraintes=2157 (incluant 1825 de coût)

Enfin, les problèmes **Small Price Only**, **Medium Price Only** et **Big Price Only** sont constitués des

TABLE 2 – Compilation de problèmes de configuration en $SLDD_+$, ADD, $SLDD_\times$ et AADD.

VCSP	$SLDD_+$		ADD	$SLDD_\times$	AADD
Instance	nœuds (arcs)	temps (s)	nœuds (arcs)	nœuds (arcs)	nœuds (arcs)
SmallPriceonly	36 (108)	< 1	4364 (7439)	3291 (7439)	36 (108)
MediumPriceonly	169 (499)	< 1	37807 (99280)	33595 (99280)	168 (495)
BigPriceonly	3317 (9687)	18	m-o	-	3317 (9687)
Small	2344 (5584)	1	299960 (637319)	14686 (33639)	2344 (5584)
Medium	6234 (17062)	6	752466 (2071474)	129803 (314648)	6234 (17062)
Big	198001 (925472)	79043	m-o	-	198001 (925472)

TABLE 3 – Compilation de réseaux bayésiens en $SLDD_+$, ADD, $SLDD_\times$ et AADD.

Réseaux bayésiens	$SLDD_\times$		ADD	$SLDD_+$	AADD
Instance	nœuds (arcs)	temps (s)	nœuds (arcs)	nœuds (arcs)	nœuds (arcs)
Cancer	13 (25)	< 1	38 (45)	23 (45)	11 (21)
Asia	23 (45)	< 1	415 (431)	216 (431)	23 (45)
Car-starts	41 (83)	< 1	42741 (64029)	19632 (39265)	38 (77)
Alarm	1301 (3993)	< 1	m-o	-	1301 (3993)
Hailfinder25	32718 (108083)	8	m-o	-	32713 (108063)

seules contraintes de coût des instances **Small**, **Medium** et **Big**, respectivement (les contraintes dures sont omises).

5.1 Efficacité spatiale des heuristiques

Pour tester l'efficacité des différents heuristiques, nous avons mesuré, pour chacune d'elles, le temps de compilation de chaque instance en SLDD, ainsi que le nombre de nœuds et d'arcs du diagramme résultant. Les problèmes de configuration avec fonction de coût ont été compilés en $SLDD_+$, les instances de réseaux bayésiens ont été compilés en $SLDD_\times$. Les résultats sont présentés à la table 1. Les notations t-o et m-o signifient respectivement *time-out* (>24h) et *out of memory* (>128Mb). Toutes les expérimentations ont été effectués sur un processeur de 800MHz.

Il apparaît que MCS-Inv est généralement l'heuristique la plus performante, tant du point de vue de la taille du diagramme généré que de celui du temps de calcul. Les résultats de MCF sont mauvais lorsqu'il n'y a que des contraintes souples (problèmes **Price Only** et réseaux bayésiens), car cette heuristique n'est intéressante qu'appliquée à des contraintes dures. Néanmoins, les performances de MCF sont supérieurs à ceux de *Band-width* (au moins pour la taille) sur les problèmes alliant contraintes souples et contraintes dures, et même supérieurs à ceux de MCS-Inv en taille sur le problème **Medium**. Notons également la faiblesse de *Force* en toutes circonstances. Au vu de ces résultats, nous avons choisi d'utiliser MCS-Inv pour la suite des expérimentations.

5.2 Efficacité spatiale des types de VDD

Nous avons voulu aussi comparer la taille des diagrammes obtenus selon les différents langages considérés dans l'article. Les jeux d'essai ont été compilés sous la forme de SLDD ($SLDD_+$ pour les VCSP et $SLDD_\times$ pour les réseaux bayésiens) puis traduits dans les autres langages. Les tables 2 et 3 indiquent les tailles obtenues pour la représentation des problèmes de configuration et des réseaux bayésiens sous la forme de ADD, $SLDD_+$, $SLDD_\times$ et AADD.

Ces expérimentations confirment les hypothèses de compacité des différents langages. En effet, on retrouve bien que le langage ADD (le moins succinct) est toujours moins compact que $SLDD_+$, $SLDD_\times$ et AADD. À l'inverse, l'AADD est toujours au moins aussi compact que $SLDD_+$, $SLDD_\times$ et ADD. Il s'avère qu'un langage offre en pratique une bonne compacité si celui-ci intègre l'opérateur adéquat au type d'instance considéré. Ainsi les langages $SLDD_+$ et AADD qui intègrent l'addition sont plus efficaces spatialement pour la compilation de VCSP dont les contraintes sont de nature additive (portant sur un prix), alors que $SLDD_\times$ et AADD sont plus compacts pour la compilation de réseaux bayésiens, où les contraintes sont de nature multiplicative (tables de probabilités) [3]. . À l'inverse, un opérateur non pertinent n'apporte que peu, voire pas, d'amélioration. Ainsi, la comparaison entre les différents langages de type SLDD et le langage AADD nous montre que l'utilisation d'un deuxième opérateur n'apporte pas, dans les problèmes purement additifs ou purement multiplicatifs, d'amélioration pratique en terme de compacité. L'utilisation d'un AADD par rapport à un $SLDD_+$ (resp. $SLDD_\times$) n'apporte pas de

3. Lors de la compilation de réseaux bayésiens en ADD et AADD, nous obtenons des représentations nettement moins succinctes que celles décrites dans [17]. Ceci s'explique par le choix que nous avons fait de représenter les valeurs réelles avec une précision plus importante. Dans le cas des ADDs, le nombre de valeurs finales possible explose clairement quand la précision augmente.

réelle amélioration pour la compilation des problèmes de configuration avec coût (resp. réseaux bayésiens)

6 Conclusion

Dans cette article nous avons étudié la compilation vers et la traduction entre différents types de diagrammes de décision valués. Du point de vue théorique le langage AADD peut être vu comme une généralisation des langages SLDD_+ et $\text{SLDD}_\times$, langages qui généralisent eux-mêmes le langage ADD : AADD est donc, parmi eux, le langage de compilation le plus succinct. Les expérimentations montrent que, dans la pratique, pour la compilation de problèmes de nature purement additive (respectivement purement multiplicative), seul l'opérateur $+$ (respectivement $\times$) est utile, et que l'utilisation de SLDD_+ (respectivement $\text{SLDD}_\times$) suffit amplement à la compilation du problème original. Nous avons également étudié plusieurs heuristiques, et vérifié que les meilleures tendaient au regroupement des variables incluses dans un même groupe de contraintes. Dans nos travaux futurs, nous prévoyons d'étudier la complexité et d'implémenter des requêtes et des transformations de VDDs, afin de pouvoir utiliser ces structures de données pour traiter efficacement (et avec une garantie de temps de réponse) des problèmes de configuration en ligne avec maintien d'un indicateur de coût minimal.

Références

[1] Fadi A. Aloul, Igor L. Markov, and Karem A. Sakallah. Force : a fast and easy-to-implement variable-ordering heuristic. In *ACM Great Lakes symposium on VLSI*, GLSVLSI '03, 2003.

[2] Jérôme Amilhastre. *Représentation par automate d'ensemble de solutions de problèmes de satisfaction de contraintes*. PhD thesis, Université de Montpellier II, 1999.

[3] Jérôme Amilhastre, Hélène Fargier, and Pierre Marquis. Consistency restoration and explanations in dynamic CSPs application to configuration. *Artif. Intell.*, 135(1-2) :199–234, 2002.

[4] R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebraic decision diagrams and their applications. In *Proc. of ICCAD'93*, pages 188–191, 1993.

[5] S. Bistarelli, U. Montanari, F. Rossi, T. Schiex, G. Verfaillie, and H. Fargier. Semiring-based csps and valued csps : Frameworks, properties, and comparison. *Constraints*, 4(3) :199–240, 1999.

[6] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35 :677–691, 1986.

[7] Fabio Gagliardi Cozman. JavaBayes Version 0.347, Bayesian Networks in Java, User Manual. Technical report, dec 2002. Benchmarks at http ://sites.poli.usp.br/pmr/ltd/ Software/javabayes/Home/node3.html.

[8] Rolf Drechsler. Evaluation of static variable ordering heuristics for mdd construction. In *ISMVL*, pages 254–260, 2002.

[9] Helene Fargier, Pierre Marquis, and Nicolas Schmidt. Semiring labelled decision diagrams, revisited : Canonicity and spatial efficiency issues, irit research report rr–2013-20–fr. Technical report, IRIT, 2013.

[10] Tarik Hadzic. A bdd-based approach to interactive configuration. In *CP*, page 797, 2004.

[11] Tarik Hadzic and Henrik Reif Andersen. A BDD-based polytime algorithm for cost-bounded interactive configuration. In *AAAI*, pages 62–67, 2006.

[12] Tarik Hadzic, Rune Jensen, and Henrik Reif Andersen. Calculating valid domains for bdd-based interactive configuration. *CoRR*, 0704.1394, 2007.

[13] Jesse Hoey, Robert St-Aubin, Alan J. Hu, and Craig Boutilier. SPUDD : Stochastic planning using decision diagrams. In *UAI*, 1999.

[14] Yung-Te Lai, Massoud Pedram, and Sarma B. K. Vrudhula. Formal verification using edge-valued binary decision diagrams. *IEEE Trans. on Computers*, 45(2) :247–255, 1996.

[15] Yung-Te Lai and Sarma Sastry. Edge-valued binary decision diagrams for multi-level hierarchical verification. In *DAC*, pages 608–613, 1992.

[16] Olivier Roussel and Christophe Lecoutre. XML Representation of Constraint Networks : Format XCSP 2.1. Technical report, CoRR abs/0902.2362, feb 2009.

[17] Scott Sanner and David A. McAllester. Affine algebraic decision diagrams (AADDs) and their application to structured probabilistic inference. In *Proc. of IJCAI'05*, pages 1384–1390, 2005.

[18] Paul Tafertshofer and Massoud Pedram. Factored edge-valued binary decision diagrams. *Formal Methods in System Design*, 10(2/3), 1997.

[19] Robert E. Tarjan and Mihalis Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13(3) :566–579, July 1984.

[20] Nic Wilson. Decision diagrams for the computation of semiring valuations. In *IJCAI*, 2005.